

Taylor Models

Christoph Traut and Fabian Immler

February 23, 2021

Abstract

We present a formally verified implementation of multivariate Taylor models. Taylor models are a form of rigorous polynomial approximation, consisting of an approximation polynomial based on Taylor expansions, combined with a rigorous bound on the approximation error. Taylor models were introduced as a tool to mitigate the dependency problem of interval arithmetic. Our implementation automatically computes Taylor models for the class of elementary functions, expressed by composition of arithmetic operations and basic functions like exp, sin, or square root.

Contents

1	Topology for Floating Point Numbers	2
2	Horner Evaluation	3
3	Splitting polynomials to reduce floating point precision	7
4	Splitting polynomials by degree	8
5	Multivariate Taylor Models	12
5.1	Computing interval bounds on arithmetic expressions	12
5.2	Definition of Taylor models and notion of rangeity	12
5.3	Interval bounds for Taylor models	13
5.4	Computing taylor models for basic, univariate functions . . .	16
5.4.1	Derivations of floatarith expressions	17
5.4.2	Computing Taylor models for arbitrary univariate expressions	18
5.5	Operations on Taylor models	20
5.6	Computing Taylor models for multivariate expressions	29
5.7	Computing bounds for floatarith expressions	32

1 Topology for Floating Point Numbers

theory *Float-Topology*

imports

HOL-Analysis.Multivariate-Analysis

HOL-Library.Float

begin

This topology is totally disconnected and not complete, in which sense is it useful? Perhaps for convergence of intervals?

unbundle *float.lifting*

instantiation *float :: dist*

begin

lift-definition *dist-float :: float \Rightarrow float \Rightarrow real is dist* *(proof)*

lemma *dist-float-eq-0-iff: (dist x y = 0) = (x = y) for x y::float*
(proof)

lemma *dist-float-triangle2: dist x y \leq dist x z + dist y z for x y z::float*
(proof)

instance *(proof)*

end

instantiation *float :: uniformity*

begin

definition *uniformity-float :: (float \times float) filter*

where *uniformity-float = (INF e \in {0<..}. principal {(x, y). dist x y < e})*

instance *(proof)*

end

lemma *float-dense-in-real:*

fixes *x :: real*

assumes *x < y*

shows $\exists r \in \text{float}. x < r \wedge r < y$

(proof)

lemma *real-of-float-dense:*

fixes *x y :: real*

assumes *x < y*

shows $\exists q :: \text{float}. x < \text{real-of-float } q \wedge \text{real-of-float } q < y$

(proof)

instantiation *float :: linorder-topology*

begin

definition *open-float::float set* \Rightarrow *bool* **where**
open-float S = $(\forall x \in S. \exists e > 0. \forall y. \text{dist } y \ x < e \longrightarrow y \in S)$

instance
 \langle *proof* \rangle

end

instance *float :: metric-space*
 \langle *proof* \rangle

instance *float::topological-ab-group-add*
 \langle *proof* \rangle

lifting-update *float.lifting*
lifting-forget *float.lifting*

end

2 Horner Evaluation

theory *Horner-Eval*
imports *HOL-Library.Interval*
begin

Function and lemmas for evaluating polynomials via the horner scheme. Because interval multiplication is not distributive, interval polynomials expressed as a sum of monomials are not equivalent to their respective horner form. The functions and lemmas in this theory can be used to express interval polynomials in horner form and prove facts about them.

fun *horner-eval'* **where**
horner-eval' f x v 0 = *v*
 $|$ *horner-eval' f x v (Suc i)* = *horner-eval' f x (f i + x * v) i*

definition *horner-eval*
where *horner-eval f x n* = *horner-eval' f x 0 n*

lemma *horner-eval-cong*:
assumes $\bigwedge i. i < n \implies f i = g i$
assumes $x = y$
assumes $n = m$
shows *horner-eval f x n = horner-eval g y m*
 \langle *proof* \rangle

lemma *horner-eval-eq-setsum*:
fixes $x::'a::\text{linordered-idom}$
shows *horner-eval f x n* = $(\sum_{i < n}. f i * x^i)$

<proof>

lemma *horner-eval-Suc[simp]*:

fixes $x::'a::\text{linordered-idom}$

shows $\text{horner-eval } f \ x \ (\text{Suc } n) = \text{horner-eval } f \ x \ n + (f \ n) * x^{\widehat{n}}$

<proof>

lemma *horner-eval-Suc'[simp]*:

fixes $x::'a::\{\text{comm-monoid-add, times}\}$

shows $\text{horner-eval } f \ x \ (\text{Suc } n) = f \ 0 + x * (\text{horner-eval } (\lambda i. f \ (\text{Suc } i)) \ x \ n)$

<proof>

lemma *horner-eval-0[simp]*:

shows $\text{horner-eval } f \ x \ 0 = 0$

<proof>

lemma *horner-eval'-interval*:

fixes $x::'a::\text{linordered-ring}$

assumes $\bigwedge i. i < n \implies f \ i \in \text{set-of } (g \ i)$

assumes $x \in_i I \ v \in_i V$

shows $\text{horner-eval}' \ f \ x \ v \ n \in_i \text{horner-eval}' \ g \ I \ V \ n$

<proof>

lemma *horner-eval-interval*:

fixes $x::'a::\text{linordered-idom}$

assumes $\bigwedge i. i < n \implies f \ i \in \text{set-of } (g \ i)$

assumes $x \in \text{set-of } I$

shows $\text{horner-eval } f \ x \ n \in_i \text{horner-eval } g \ I \ n$

<proof>

end

theory *Polynomial-Expression-Additional*

imports

Polynomial-Expression

HOL-Decision-Procs.Approximation

begin

lemma *real-of-float-eq-zero-iff[simp]*: $\text{real-of-float } x = 0 \longleftrightarrow x = 0$

<proof>

Theory *Taylor-Models.Polynomial-Expression* contains a, more or less, 1:1 generalization of theory *Multivariate-Polynomial*. Any additions belong here.

declare $[[\text{coercion-map } \text{map-poly}]]$

declare $[[\text{coercion } \text{interval-of}::\text{float} \Rightarrow \text{float } \text{interval}]]$

Apply float interval arguments to a float poly.

value *Ipoly* $[\text{Ivl } (\text{Float } 4 \ (-6)) \ (\text{Float } 10 \ 6)] \ (\text{poly.Add } (\text{poly.C } (\text{Float } 3 \ 5)) \ (\text{poly.Bound } 0))$

map-poly for homomorphisms

lemma *map-poly-homo-polyadd-eq-zero-iff*:

$$\text{map-poly } f (p +_p q) = 0_p \longleftrightarrow p +_p q = 0_p$$

$$\text{if } [\text{symmetric}, \text{simp}]: \bigwedge x y. f (x + y) = f x + f y \bigwedge x. f x = 0 \longleftrightarrow x = 0$$

<proof>

lemma *zero-iffD*: $(\bigwedge x. f x = 0 \longleftrightarrow x = 0) \implies f 0 = 0$

<proof>

lemma *map-poly-homo-polyadd*:

$$\text{map-poly } f (p1 +_p p2) = \text{map-poly } f p1 +_p \text{map-poly } f p2$$

$$\text{if } [\text{simp}]: \bigwedge x y. f (x + y) = f x + f y \bigwedge x. f x = 0 \longleftrightarrow x = 0$$

<proof>

lemma *map-poly-homo-polyneg*:

$$\text{map-poly } f (\sim_p p1) = \sim_p (\text{map-poly } f p1)$$

$$\text{if } [\text{simp}]: \bigwedge x y. f (- x) = - f x$$

<proof>

lemma *map-poly-homo-polysub*:

$$\text{map-poly } f (p1 -_p p2) = \text{map-poly } f p1 -_p \text{map-poly } f p2$$

$$\text{if } [\text{simp}]: \bigwedge x y. f (x + y) = f x + f y \bigwedge x. f x = 0 \longleftrightarrow x = 0 \bigwedge x y. f (- x) = - f x$$

<proof>

lemma *map-poly-homo-polymul*:

$$\text{map-poly } f (p1 *_p p2) = \text{map-poly } f p1 *_p \text{map-poly } f p2$$

$$\text{if } [\text{simp}]: \bigwedge x y. f (x + y) = f x + f y \bigwedge x. f x = 0 \longleftrightarrow x = 0 \bigwedge x y. f (x * y) = f x * f y$$

<proof>

lemma *map-poly-homo-polypow*:

$$\text{map-poly } f (p1 \widehat{ }_p n) = \text{map-poly } f p1 \widehat{ }_p n$$

$$\text{if } [\text{simp}]: \bigwedge x y. f (x + y) = f x + f y \bigwedge x. f x = 0 \longleftrightarrow x = 0 \bigwedge x y. f (x * y) = f x * f y$$

$$f 1 = 1$$

<proof>

lemmas *map-poly-homo-polyarith* = *map-poly-homo-polyadd* *map-poly-homo-polyneg*
map-poly-homo-polysub *map-poly-homo-polymul* *map-poly-homo-polypow*

Count the number of parameters of a polynomial.

fun *num-params* :: 'a poly \Rightarrow nat

where *num-params* (poly.C c) = 0

$$| \text{num-params } (\text{poly.Bound } n) = \text{Suc } n$$

$$| \text{num-params } (\text{poly.Add } a b) = \max (\text{num-params } a) (\text{num-params } b)$$

$$| \text{num-params } (\text{poly.Sub } a b) = \max (\text{num-params } a) (\text{num-params } b)$$

$$| \text{num-params } (\text{poly.Mul } a b) = \max (\text{num-params } a) (\text{num-params } b)$$

$$| \text{num-params } (\text{poly.Neg } a) = \text{num-params } a$$

$| \text{num-params } (\text{poly.Pw } a \ n) = \text{num-params } a$
 $| \text{num-params } (\text{poly.CN } a \ n \ b) = \max (\max (\text{num-params } a) (\text{num-params } b))$
(Suc n)

lemma *num-params-map-poly[simp]*:
shows $\text{num-params } (\text{map-poly } f \ p) = \text{num-params } p$
<proof>

lemma *num-params-polyadd*:
shows $\text{num-params } (p1 +_p p2) \leq \max (\text{num-params } p1) (\text{num-params } p2)$
<proof>

lemma *num-params-polyneg*:
shows $\text{num-params } (\sim_p p) = \text{num-params } p$
<proof>

lemma *num-params-polymul*:
shows $\text{num-params } (p1 *_p p2) \leq \max (\text{num-params } p1) (\text{num-params } p2)$
<proof>

lemma *num-params-polypow*:
shows $\text{num-params } (p \hat{^}_p n) \leq \text{num-params } p$
<proof>

lemma *num-params-polynate*:
shows $\text{num-params } (\text{polynate } p) \leq \text{num-params } p$
<proof>

lemma *polynate-map-poly-real[simp]*:
fixes $p :: \text{float poly}$
shows $\text{map-poly real-of-float } (\text{polynate } p) = \text{polynate } (\text{map-poly real-of-float } p)$
<proof>

Evaluating a float poly is equivalent to evaluating the corresponding real poly with the float parameters converted to reals.

lemma *Ipoly-real-float-equiv*:
fixes $p :: \text{float poly}$ **and** $xs :: \text{float list}$
assumes $\text{num-params } p \leq \text{length } xs$
shows $\text{Ipoly } xs (p :: \text{real poly}) = \text{Ipoly } xs \ p$
<proof>

Evaluating an 'a poly with 'a interval arguments is monotone.

lemma *Ipoly-interval-args-mono*:
fixes $p :: 'a :: \text{linordered-idom poly}$
and $x :: 'a \text{ list}$
and $xs :: 'a \text{ interval list}$
assumes $x \text{ all-in}_i \ xs$
assumes $\text{num-params } p \leq \text{length } xs$
shows $\text{Ipoly } x \ p \in \text{set-of } (\text{Ipoly } xs (\text{map-poly interval-of } p))$

<proof>

lemma *Ipoly-interval-args-inc-mono*:

fixes $p::'a::\{\text{real-normed-algebra, linear-continuum-topology, linordered-idom}\}$ *poly*
and $I::'a$ *interval list* **and** $J::'a$ *interval list*
assumes $\text{num-params } p \leq \text{length } I$
assumes I *all-subset* J
shows $\text{set-of } (I\text{poly } I \text{ (map-poly interval-of } p)) \subseteq \text{set-of } (I\text{poly } J \text{ (map-poly interval-of } p))$
<proof>

3 Splitting polynomials to reduce floating point precision

TODO: Move this! Definitions regarding floating point numbers should not be in a theory about polynomials.

fun *float-prec* :: *float* \Rightarrow *int*
where *float-prec* $f = (\text{let } p = \text{exponent } f \text{ in if } p \geq 0 \text{ then } 0 \text{ else } -p)$

fun *float-round* :: *nat* \Rightarrow *float* \Rightarrow *float*
where *float-round prec* $f = (\text{let } d = \text{float-down } \text{prec } f; u = \text{float-up } \text{prec } f$
 $\text{in if } f - d < u - f \text{ then } d \text{ else } u)$

Splits any polynomial p into two polynomials l, r , such that $\forall x::\text{real. } p(x) = l(x) + r(x)$ and all floating point coefficients in p are rounded to precision prec . Not all cases need to give good results. Polynomials normalized with *polynat* only contain *poly.C* and *poly.CN* constructors.

fun *split-by-prec* :: *nat* \Rightarrow *float poly* \Rightarrow *float poly* * *float poly*
where *split-by-prec prec* (*poly.C* f) = ($\text{let } r = \text{float-round } \text{prec } f \text{ in } (\text{poly.C } r, \text{poly.C } (f - r))$)
| *split-by-prec prec* (*poly.Bound* n) = (*poly.Bound* $n, \text{poly.C } 0$)
| *split-by-prec prec* (*poly.Add* $l r$) = ($\text{let } (ll, lr) = \text{split-by-prec } \text{prec } l;$
 $(rl, rr) = \text{split-by-prec } \text{prec } r$
 $\text{in } (\text{poly.Add } ll \ rl, \text{poly.Add } lr \ rr)$)
| *split-by-prec prec* (*poly.Sub* $l r$) = ($\text{let } (ll, lr) = \text{split-by-prec } \text{prec } l;$
 $(rl, rr) = \text{split-by-prec } \text{prec } r$
 $\text{in } (\text{poly.Sub } ll \ rl, \text{poly.Sub } lr \ rr)$)
| *split-by-prec prec* (*poly.Mul* $l r$) = ($\text{let } (ll, lr) = \text{split-by-prec } \text{prec } l;$
 $(rl, rr) = \text{split-by-prec } \text{prec } r$
 $\text{in } (\text{poly.Mul } ll \ rl, \text{poly.Add } (\text{poly.Add } (\text{poly.Mul } lr \ rl) (\text{poly.Mul } ll \ rr)) (\text{poly.Mul } lr \ rr))$)
| *split-by-prec prec* (*poly.Neg* p) = ($\text{let } (l, r) = \text{split-by-prec } \text{prec } p \text{ in } (\text{poly.Neg } l, \text{poly.Neg } r)$)
| *split-by-prec prec* (*poly.Pw* p 0) = (*poly.C* $1, \text{poly.C } 0$)
| *split-by-prec prec* (*poly.Pw* p (*Suc* n)) = ($\text{let } (l, r) = \text{split-by-prec } \text{prec } p \text{ in } (\text{poly.Pw } l \ n, \text{poly.Sub } (\text{poly.Pw } p \text{ (Suc } n)) (\text{poly.Pw } l \ n))$)

| *split-by-prec prec (poly.CN c n p) = (let (cl, cr) = split-by-prec prec c;
 (pl, pr) = split-by-prec prec p
 in (poly.CN cl n pl, poly.CN cr n pr))*

TODO: Prove precision constraint on *l*.

lemma *split-by-prec-correct*:
fixes *args* :: *real list*
assumes $(l, r) = \text{split-by-prec } \text{prec } p$
shows $I\text{poly } \text{args } p = I\text{poly } \text{args } l + I\text{poly } \text{args } r$ (**is** ?P1)
and $\text{num-params } l \leq \text{num-params } p$ (**is** ?P2)
and $\text{num-params } r \leq \text{num-params } p$ (**is** ?P3)
 <proof>

4 Splitting polynomials by degree

fun *maxdegree* :: ('a::zero) *poly* \Rightarrow *nat*
where *maxdegree* (poly.C c) = 0
 | *maxdegree* (poly.Bound n) = 1
 | *maxdegree* (poly.Add l r) = max (*maxdegree* l) (*maxdegree* r)
 | *maxdegree* (poly.Sub l r) = max (*maxdegree* l) (*maxdegree* r)
 | *maxdegree* (poly.Mul l r) = *maxdegree* l + *maxdegree* r
 | *maxdegree* (poly.Neg p) = *maxdegree* p
 | *maxdegree* (poly.Pw p n) = n * *maxdegree* p
 | *maxdegree* (poly.CN c n p) = max (*maxdegree* c) (1 + *maxdegree* p)

fun *split-by-degree* :: *nat* \Rightarrow 'a::zero *poly* \Rightarrow 'a *poly* * 'a *poly*
where *split-by-degree* n (poly.C c) = (poly.C c, poly.C 0)
 | *split-by-degree* 0 p = (poly.C 0, p)
 | *split-by-degree* (Suc n) (poly.CN c v p) = (
 let (cl, cr) = *split-by-degree* (Suc n) c;
 (pl, pr) = *split-by-degree* n p
 in (poly.CN cl v pl, poly.CN cr v pr))

— This function is only intended for use on polynomials in normal form. Hence most cases never get executed.

| *split-by-degree* n p = (poly.C 0, p)

lemma *split-by-degree-correct*:
fixes *x* :: *real list* **and** *p* :: *float poly*
assumes $(l, r) = \text{split-by-degree } \text{ord } p$
shows $\text{maxdegree } l \leq \text{ord}$ (**is** ?P1)
and $I\text{poly } x p = I\text{poly } x l + I\text{poly } x r$ (**is** ?P2)
and $\text{num-params } l \leq \text{num-params } p$ (**is** ?P3)
and $\text{num-params } r \leq \text{num-params } p$ (**is** ?P4)
 <proof>

Operations on lists.

lemma *length-map2[simp]*: $\text{length } (\text{map2 } f a b) = \min (\text{length } a) (\text{length } b)$
 <proof>


```

lemma map2-nth[simp]:
  assumes  $n < \text{length } a$ 
  assumes  $n < \text{length } b$ 
  shows  $(\text{map2 } f \ a \ b)!n = f \ (a!n) \ (b!n)$ 
  <proof>

```

Translating a polynomial by a vector.

```

fun poly-translate :: 'a list  $\Rightarrow$  'a poly  $\Rightarrow$  'a poly
  where poly-translate vs (poly.C c) = poly.C c
  | poly-translate vs (poly.Bound n) = poly.Add (poly.Bound n) (poly.C (vs ! n))
  | poly-translate vs (poly.Add l r) = poly.Add (poly-translate vs l) (poly-translate vs r)
  | poly-translate vs (poly.Sub l r) = poly.Sub (poly-translate vs l) (poly-translate vs r)
  | poly-translate vs (poly.Mul l r) = poly.Mul (poly-translate vs l) (poly-translate vs r)
  | poly-translate vs (poly.Neg p) = poly.Neg (poly-translate vs p)
  | poly-translate vs (poly.Pw p n) = poly.Pw (poly-translate vs p) n
  | poly-translate vs (poly.CN c n p) = poly.Add (poly-translate vs c) (poly.Mul (poly.Add (poly.Bound n) (poly.C (vs ! n))) (poly-translate vs p))

```

Translating a polynomial is equivalent to translating its argument.

```

lemma poly-translate-correct:
  assumes  $\text{num-params } p \leq \text{length } x$ 
  assumes  $\text{length } x = \text{length } v$ 
  shows  $\text{Ipoly } x \ (\text{poly-translate } v \ p) = \text{Ipoly } (\text{map2 } (+) \ x \ v) \ p$ 
  <proof>

```

```

lemma real-poly-translate:
  assumes  $\text{num-params } p \leq \text{length } v$ 
  shows  $\text{Ipoly } x \ (\text{map-poly real-of-float } (\text{poly-translate } v \ p)) = \text{Ipoly } x \ (\text{poly-translate } v \ (\text{map-poly real-of-float } p))$ 
  <proof>

```

```

lemma num-params-poly-translate[simp]:
  shows  $\text{num-params } (\text{poly-translate } v \ p) = \text{num-params } p$ 
  <proof>

```

end

theory Taylor-Models-Misc

imports

HOL-Library.Float

HOL-Library.Function-Algebras

HOL-Decision-Procs.Approximation

Affine-Arithmetic.Floatarith-Expression

begin

This theory contains anything that doesn't belong anywhere else.

lemma *of-nat-real-float-equiv*: $(\text{of-nat } n :: \text{real}) = (\text{of-nat } n :: \text{float})$
<proof>

lemma *fact-real-float-equiv*: $(\text{fact } n :: \text{float}) = (\text{fact } n :: \text{real})$
<proof>

lemma *Some-those-length*:
 $\text{those } ys = \text{Some } xs \implies \text{length } xs = \text{length } ys$
<proof>

lemma *those-eq-None-iff*: $\text{those } ys = \text{None} \iff \text{None} \in \text{set } ys$
<proof>

lemma *those-eq-Some-iff*: $\text{those } ys = (\text{Some } xs) \iff (ys = \text{map } \text{Some } xs)$
<proof>

lemma *Some-those-nth*:
assumes $\text{those } ys = \text{Some } xs$
assumes $i < \text{length } xs$
shows $\text{Some } (xs!i) = ys!i$
<proof>

lemma *fun-pow*: $f^{\wedge}n = (\lambda x. (f x)^{\wedge}n)$
<proof>

context includes *floatarith-notation begin*

Translate floatarith expressions by a vector of floats.

fun *fa-translate* :: *float list* \Rightarrow *floatarith* \Rightarrow *floatarith*
where *fa-translate* v (*Add* a b) = *Add* (*fa-translate* v a) (*fa-translate* v b)
| *fa-translate* v (*Minus* a) = *Minus* (*fa-translate* v a)
| *fa-translate* v (*Mult* a b) = *Mult* (*fa-translate* v a) (*fa-translate* v b)
| *fa-translate* v (*Inverse* a) = *Inverse* (*fa-translate* v a)
| *fa-translate* v (*Cos* a) = *Cos* (*fa-translate* v a)
| *fa-translate* v (*Arctan* a) = *Arctan* (*fa-translate* v a)
| *fa-translate* v (*Min* a b) = *Min* (*fa-translate* v a) (*fa-translate* v b)
| *fa-translate* v (*Max* a b) = *Max* (*fa-translate* v a) (*fa-translate* v b)
| *fa-translate* v (*Abs* a) = *Abs* (*fa-translate* v a)
| *fa-translate* v (*Sqrt* a) = *Sqrt* (*fa-translate* v a)
| *fa-translate* v (*Exp* a) = *Exp* (*fa-translate* v a)
| *fa-translate* v (*Ln* a) = *Ln* (*fa-translate* v a)
| *fa-translate* v (*Var* n) = *Add* (*Var* n) (*Num* ($v!n$))
| *fa-translate* v (*Power* a n) = *Power* (*fa-translate* v a) n
| *fa-translate* v (*Powr* a b) = *Powr* (*fa-translate* v a) (*fa-translate* v b)
| *fa-translate* v (*Floor* x) = *Floor* (*fa-translate* v x)
| *fa-translate* v (*Num* c) = *Num* c
| *fa-translate* v *Pi* = *Pi*

lemma *fa-translate-correct*:

assumes *max-Var-floatarith* $f \leq \text{length } I$
assumes *length* $v = \text{length } I$
shows *interpret-floatarith* (*fa-translate* $v f$) $I = \text{interpret-floatarith } f (\text{map2 } (+)$
 $I v)$
 ⟨*proof*⟩

primrec *vars-floatarith* **where**

vars-floatarith (*Add* $a b$) = (*vars-floatarith* a) \cup (*vars-floatarith* b)
 | *vars-floatarith* (*Mult* $a b$) = (*vars-floatarith* a) \cup (*vars-floatarith* b)
 | *vars-floatarith* (*Inverse* a) = *vars-floatarith* a
 | *vars-floatarith* (*Minus* a) = *vars-floatarith* a
 | *vars-floatarith* (*Num* a) = {}
 | *vars-floatarith* (*Var* i) = { i }
 | *vars-floatarith* (*Cos* a) = *vars-floatarith* a
 | *vars-floatarith* (*Arctan* a) = *vars-floatarith* a
 | *vars-floatarith* (*Abs* a) = *vars-floatarith* a
 | *vars-floatarith* (*Max* $a b$) = (*vars-floatarith* a) \cup (*vars-floatarith* b)
 | *vars-floatarith* (*Min* $a b$) = (*vars-floatarith* a) \cup (*vars-floatarith* b)
 | *vars-floatarith* (*Pi*) = {}
 | *vars-floatarith* (*Sqrt* a) = *vars-floatarith* a
 | *vars-floatarith* (*Exp* a) = *vars-floatarith* a
 | *vars-floatarith* (*Powr* $a b$) = (*vars-floatarith* a) \cup (*vars-floatarith* b)
 | *vars-floatarith* (*Ln* a) = *vars-floatarith* a
 | *vars-floatarith* (*Power* $a n$) = *vars-floatarith* a
 | *vars-floatarith* (*Floor* a) = *vars-floatarith* a

lemma *finite-vars-floatarith[simp]*: *finite* (*vars-floatarith* x)
 ⟨*proof*⟩

end

lemma *max-Var-floatarith-eq-Max-vars-floatarith*:

max-Var-floatarith $fa = (\text{if } \text{vars-floatarith } fa = \{\} \text{ then } 0 \text{ else } \text{Suc } (\text{Max } (\text{vars-floatarith } fa)))$
 ⟨*proof*⟩

end

theory *Taylor-Models*

imports

Horner-Eval
Polynomial-Expression-Additional
Taylor-Models-Misc
HOL-Decision-Procs.Approximation
HOL-Library.Function-Algebras
HOL-Library.Set-Algebras
Affine-Arithmetic.Straight-Line-Program
Affine-Arithmetic.Affine-Approximation

begin

TODO: get rid of float poly/float interval and use real poly/real interval and

data refinement?

5 Multivariate Taylor Models

5.1 Computing interval bounds on arithmetic expressions

This is a wrapper around the "approx" function. It computes range bounds on floatarith expressions.

fun *compute-bound-fa* :: *nat* \Rightarrow *floatarith* \Rightarrow *float interval list* \Rightarrow *float interval option*
where *compute-bound-fa prec f I* = *approx prec f (map Some I)*

lemma *compute-bound-fa-correct*:
interpret-floatarith f i \in_r ivl
if *compute-bound-fa prec f I* = *Some ivl*
i all-in I
for *i::real list*
 <proof>

5.2 Definition of Taylor models and notion of rangeity

Taylor models are a pair of a polynomial and an absolute error bound.

datatype *taylor-model* = *TaylorModel (tm-poly: float poly) (tm-bound: float interval)*

Taylor model for a real valuation of variables

primrec *insertion* :: (*nat* \Rightarrow 'a) \Rightarrow 'a *poly* \Rightarrow 'a::{*plus,zero,minus,uminus,times,one,power*}
where

insertion bs (C c) = *c*
 | *insertion bs (poly.Bound n)* = *bs n*
 | *insertion bs (Neg a)* = - *insertion bs a*
 | *insertion bs (poly.Add a b)* = *insertion bs a* + *insertion bs b*
 | *insertion bs (Sub a b)* = *insertion bs a* - *insertion bs b*
 | *insertion bs (Mul a b)* = *insertion bs a* * *insertion bs b*
 | *insertion bs (Pw t n)* = *insertion bs t* \wedge^n
 | *insertion bs (CN c n p)* = *insertion bs c* + (*bs n*) * *insertion bs p*

definition *range-tm* :: (*nat* \Rightarrow *real*) \Rightarrow *taylor-model* \Rightarrow *real interval* **where**
range-tm e tm = *interval-of (insertion e (tm-poly tm))* + *real-interval (tm-bound tm)*

lemma *Ipoly-num-params-cong*: *Ipoly xs p* = *Ipoly ys p*
if $\bigwedge i. i < \text{num-params } p \implies xs ! i = ys ! i$
 <proof>

lemma *insertion-num-params-cong*: *insertion e p* = *insertion f p*
if $\bigwedge i. i < \text{num-params } p \implies e i = f i$

<proof>

lemma *insertion-eq-IpolyI*: *insertion xs p = Ipoly ys p*
if $\bigwedge i. i < \text{num-params } p \implies xs \ i = ys \ ! \ i$
<proof>

lemma *Ipoly-eq-insertionI*: *Ipoly ys p = insertion xs p*
if $\bigwedge i. i < \text{num-params } p \implies xs \ i = ys \ ! \ i$
<proof>

lemma *range-tmI*:
 $x \in_i \text{range-tm } e \ tm$
if $x \in_i \text{interval-of } (\text{insertion } e \ ((\text{tm-poly } tm))) + \text{real-interval } (\text{tm-bound } tm)$
for $e::\text{nat} \Rightarrow \text{real}$
<proof>

lemma *range-tmD*:
 $x \in_i \text{interval-of } (\text{insertion } e \ (\text{tm-poly } tm)) + \text{real-interval } (\text{tm-bound } tm)$
if $x \in_i \text{range-tm } e \ tm$
for $e::\text{nat} \Rightarrow \text{real}$
<proof>

5.3 Interval bounds for Taylor models

Bound a polynomial by simply approximating it with interval arguments.

fun *compute-bound-poly* :: $\text{nat} \Rightarrow \text{float interval poly} \Rightarrow (\text{float interval list}) \Rightarrow (\text{float interval list}) \Rightarrow \text{float interval}$ **where**
 compute-bound-poly prec (poly.C f) I a = f
 | *compute-bound-poly prec (poly.Bound n) I a = round-interval prec (I ! n - (a ! n))*
 | *compute-bound-poly prec (poly.Add p q) I a =*
 round-interval prec (compute-bound-poly prec p I a + compute-bound-poly prec q I a)
 | *compute-bound-poly prec (poly.Sub p q) I a =*
 round-interval prec (compute-bound-poly prec p I a - compute-bound-poly prec q I a)
 | *compute-bound-poly prec (poly.Mul p q) I a =*
 mult-float-interval prec (compute-bound-poly prec p I a) (compute-bound-poly prec q I a)
 | *compute-bound-poly prec (poly.Neg p) I a = -compute-bound-poly prec p I a*
 | *compute-bound-poly prec (poly.Pw p n) I a = power-float-interval prec n (compute-bound-poly prec p I a)*
 | *compute-bound-poly prec (poly.CN p n q) I a =*
 round-interval prec (compute-bound-poly prec p I a +
 mult-float-interval prec (round-interval prec (I ! n - (a ! n))) (compute-bound-poly prec q I a))

Bounds on Taylor models are simply a bound on its polynomial, widened by the approximation error.

fun *compute-bound-tm* :: nat \Rightarrow float interval list \Rightarrow float interval list \Rightarrow TaylorModel \Rightarrow float interval
where *compute-bound-tm* prec I a (TaylorModel p e) = *compute-bound-poly* prec p I a + e

lemma *compute-bound-tm-def*:
compute-bound-tm prec I a tm = *compute-bound-poly* prec (tm-poly tm) I a + (tm-bound tm)
 <proof>

lemma *real-of-float-in-real-interval-of*[intro, simp]: real-of-float $x \in_r X$ **if** $x \in_i X$
 <proof>

lemma *in-set-of-round-interval*[intro, simp]:
 $x \in_r \text{round-interval } prec X$ **if** $x \in_r X$
 <proof>

lemma *in-set-real-minus-interval*[intro, simp]:
 $x - y \in_r X - Y$ **if** $x \in_r X$ $y \in_r Y$
 <proof>

lemma *real-interval-plus*: real-interval (a + b) = real-interval a + real-interval b
 <proof>

lemma *real-interval-uminus*: real-interval (- b) = - real-interval b
 <proof>

lemma *real-interval-of*: real-interval (interval-of b) = interval-of b
 <proof>

lemma *real-interval-minus*: real-interval (a - b) = real-interval a - real-interval b
 <proof>

lemma *in-set-real-plus-interval*[intro, simp]:
 $x + y \in_r X + Y$ **if** $x \in_r X$ $y \in_r Y$
 <proof>

lemma *in-set-neg-plus-interval*[intro, simp]:
 $-y \in_r -Y$ **if** $y \in_r Y$
 <proof>

lemma *in-set-real-times-interval*[intro, simp]:
 $x * y \in_r X * Y$ **if** $x \in_r X$ $y \in_r Y$
 <proof>

lemma *real-interval-one*: real-interval 1 = 1
 <proof>

lemma *real-interval-zero*: *real-interval* 0 = 0
 ⟨*proof*⟩

lemma *real-interval-power*: *real-interval* (a ^ b) = *real-interval* a ^ b
 ⟨*proof*⟩

lemma *in-set-real-power-interval*[*intro, simp*]:
 $x \wedge n \in_r X \wedge n$ **if** $x \in_r X$
 ⟨*proof*⟩

lemma *power-float-interval-real-interval*[*intro, simp*]:
 $x \wedge n \in_r \text{power-float-interval prec } n X$ **if** $x \in_r X$
 ⟨*proof*⟩

lemma *in-set-mult-float-interval*[*intro, simp*]:
 $x * y \in_r \text{mult-float-interval prec } X Y$ **if** $x \in_r X$ $y \in_r Y$
 ⟨*proof*⟩

lemma *in-set-real-minus-swap**I*: $e \ i \in_r I ! i - a ! i$
if $x - e \ i \in_r a ! i$ $x \in_r I ! i$
 ⟨*proof*⟩

definition *develops-at-within*::(*nat* \Rightarrow *real*) \Rightarrow *float interval list* \Rightarrow *float interval list* \Rightarrow *bool*
where *develops-at-within* $e \ a \ I \longleftrightarrow (a \text{ all-subset } I) \wedge (\forall i < \text{length } I. e \ i \in_r I ! i - a ! i)$

lemma *develops-at-withinI*:
assumes *all-in*: $a \text{ all-subset } I$
assumes $e: \bigwedge i. i < \text{length } I \implies e \ i \in_r I ! i - a ! i$
shows *develops-at-within* $e \ a \ I$
 ⟨*proof*⟩

lemma *develops-at-withinD*:
assumes *develops-at-within* $e \ a \ I$
shows $a \text{ all-subset } I$
 $\bigwedge i. i < \text{length } I \implies e \ i \in_r I ! i - a ! i$
 ⟨*proof*⟩

lemma *compute-bound-poly-correct*:
fixes $p::\text{float poly}$
assumes *num-params* $p \leq \text{length } I$
assumes *dev*: *develops-at-within* $e \ a \ I$
shows *insertion* $e \ (p::\text{real poly}) \in_r \text{compute-bound-poly prec } (\text{map-poly interval-of } p) \ I \ a$
 ⟨*proof*⟩

lemma *compute-bound-tm-correct*:
fixes $I :: \text{float interval list}$ **and** $f :: \text{real list} \Rightarrow \text{real}$

assumes n : $\text{num-params } (tm\text{-poly } t) \leq \text{length } I$
assumes dev : $\text{develops-at-within } e \ a \ I$
assumes $x0$: $x0 \in_i \text{range-tm } e \ t$
shows $x0 \in_r \text{compute-bound-tm prec } I \ a \ t$
 $\langle \text{proof} \rangle$

lemma *compute-bound-tm-correct-subset*:
fixes $I :: \text{float interval list}$ **and** $f :: \text{real list} \Rightarrow \text{real}$
assumes n : $\text{num-params } (tm\text{-poly } t) \leq \text{length } I$
assumes dev : $\text{develops-at-within } e \ a \ I$
shows $\text{set-of } (\text{range-tm } e \ t) \subseteq \text{set-of } (\text{real-interval } (\text{compute-bound-tm prec } I \ a \ t))$
 $\langle \text{proof} \rangle$

lemma *compute-bound-poly-mono*:
assumes $\text{num-params } p \leq \text{length } I$
assumes mem : $I \text{ all-subset } J \ a \ \text{all-subset } I$
shows $\text{set-of } (\text{compute-bound-poly prec } p \ I \ a) \subseteq \text{set-of } (\text{compute-bound-poly prec } p \ J \ a)$
 $\langle \text{proof} \rangle$

lemma *compute-bound-tm-mono*:
fixes $I :: \text{float interval list}$ **and** $f :: \text{real list} \Rightarrow \text{real}$
assumes $\text{num-params } (tm\text{-poly } t) \leq \text{length } I$
assumes $I \text{ all-subset } J$
assumes $a \text{ all-subset } I$
shows $\text{set-of } (\text{compute-bound-tm prec } I \ a \ t) \subseteq \text{set-of } (\text{compute-bound-tm prec } J \ a \ t)$
 $\langle \text{proof} \rangle$

5.4 Computing taylor models for basic, univariate functions

definition *tm-const* :: $\text{float} \Rightarrow \text{taylor-model}$
where $tm\text{-const } c = \text{TaylorModel } (\text{poly.C } c) \ 0$

context includes *floatarith-notation* **begin**

definition *tm-pi* :: $\text{nat} \Rightarrow \text{taylor-model}$
where $tm\text{-pi } prec =$ (
 $\text{let } pi\text{-ivl} = \text{the } (\text{compute-bound-fa prec } Pi \ [])$
 $\text{in } \text{TaylorModel } (\text{poly.C } (\text{mid } pi\text{-ivl})) \ (\text{centered } pi\text{-ivl})$
 $)$

lemma *zero-real-interval[intro,simp]*: $0 \in_r \ 0$
 $\langle \text{proof} \rangle$

lemma *range-TM-tm-const[simp]*: $\text{range-tm } e \ (tm\text{-const } c) = \text{interval-of } c$
 $\langle \text{proof} \rangle$

lemma *num-params-tm-const[simp]*: $\text{num-params (tm-poly (tm-const c))} = 0$
 ⟨proof⟩

lemma *num-params-tm-pi[simp]*: $\text{num-params (tm-poly (tm-pi prec))} = 0$
 ⟨proof⟩

lemma *range-tm-tm-pi*: $\text{pi} \in_i \text{range-tm } e \text{ (tm-pi prec)}$
 ⟨proof⟩

5.4.1 Derivations of floatarith expressions

Compute the nth derivative of a floatarith expression

fun *deriv* :: $\text{nat} \Rightarrow \text{floatarith} \Rightarrow \text{nat} \Rightarrow \text{floatarith}$
where $\text{deriv } v \ f \ 0 = f$
 | $\text{deriv } v \ f \ (\text{Suc } n) = \text{DERIV-floatarith } v \ (\text{deriv } v \ f \ n)$

lemma *isDERIV-DERIV-floatarith*:
assumes $\text{isDERIV } v \ f \ vs$
shows $\text{isDERIV } v \ (\text{DERIV-floatarith } v \ f) \ vs$
 ⟨proof⟩

lemma *isDERIV-is-analytic*:
 $\text{isDERIV } i \ (\text{Taylor-Models.deriv } i \ f \ n) \ xs$
if $\text{isDERIV } i \ f \ xs$
 ⟨proof⟩

lemma *deriv-correct*:
assumes $\text{isDERIV } i \ f \ (xs[i:=t]) \ i < \text{length } xs$
shows $((\lambda x. \text{interpret-floatarith } (\text{deriv } i \ f \ n) \ (xs[i:=x])) \text{ has-real-derivative } \text{interpret-floatarith } (\text{deriv } i \ f \ (\text{Suc } n)) \ (xs[i:=t]))$
 (at t within S)
 ⟨proof⟩

Faster derivation for univariate functions, producing smaller terms and thus less over-approximation.

TODO: Extend to Arctan, Log!

fun *deriv-rec* :: $\text{floatarith} \Rightarrow \text{nat} \Rightarrow \text{floatarith}$
where $\text{deriv-rec } (\text{Exp } (\text{Var } 0)) \ n = \text{Exp } (\text{Var } 0)$
 | $\text{deriv-rec } (\text{Cos } (\text{Var } 0)) \ n = (\text{case } n \ \text{mod } 4$
 of $0 \Rightarrow \text{Cos } (\text{Var } 0)$
 | $\text{Suc } 0 \Rightarrow \text{Minus } (\text{Sin } (\text{Var } 0))$
 | $\text{Suc } (\text{Suc } 0) \Rightarrow \text{Minus } (\text{Cos } (\text{Var } 0))$
 | $\text{Suc } (\text{Suc } (\text{Suc } 0)) \Rightarrow \text{Sin } (\text{Var } 0)$
 | $\text{deriv-rec } (\text{Inverse } (\text{Var } 0)) \ n = (\text{if } n = 0 \ \text{then } \text{Inverse } (\text{Var } 0) \ \text{else } \text{Mult } (\text{Num } (\text{fact } n * (\text{if } n \ \text{mod } 2 = 0 \ \text{then } 1 \ \text{else } -1))) \ (\text{Inverse } (\text{Power } (\text{Var } 0) (\text{Suc } n))))$
 | $\text{deriv-rec } f \ n = \text{deriv } 0 \ f \ n$

lemma *deriv-rec-correct*:

assumes *isDERIV* 0 *f* (*xs*[0:=*t*]) 0 < length *xs*
shows ((λx . *interpret-floatarith* (*deriv-rec* *f* *n*) (*xs*[0:=*x*])) *has-real-derivative*
interpret-floatarith (*deriv-rec* *f* (*Suc* *n*)) (*xs*[0:=*t*])) (*at t within S*)
 ⟨*proof*⟩

lemma *deriv-rec-0-idem[simp]*:
shows *deriv-rec* *f* 0 = *f*
 ⟨*proof*⟩

5.4.2 Computing Taylor models for arbitrary univariate expressions

fun *tmf-c* :: *nat* ⇒ *float interval list* ⇒ *floatarith* ⇒ *nat* ⇒ *float interval option*
where *tmf-c prec I f i* = *compute-bound-fa prec* (*Mult* (*deriv-rec* *f* *i*) (*Inverse*
 (*Num* (*fact* *i*)))) *I*
 — The interval coefficients of the Taylor polynomial, i.e. the real coefficients approximated by a float interval.

fun *tmf-ivl-cs* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float list* ⇒ *floatarith* ⇒ *float interval list option*
where *tmf-ivl-cs prec ord I a f* = *those* (*map* (*tmf-c prec a f*) [0..*ord*] @ [*tmf-c prec I f ord*])
 — Make a list of bounds on the *n*+1 coefficients, with the *n*+1-th coefficient bounding the remainder term of the Taylor-Lagrange formula.

fun *tmf-polys* :: *float interval list* ⇒ *float poly* × *float interval poly*
where *tmf-polys []* = (*poly.C* 0, *poly.C* 0)
 | *tmf-polys (c # cs)* = (
 let (*pf*, *pi*) = *tmf-polys cs*
 in (*poly.CN* (*poly.C* (*mid* *c*)) 0 *pf*, *poly.CN* (*poly.C* (*centered* *c*)) 0 *pi*)
)

fun *tm-floatarith* :: *nat* ⇒ *nat* ⇒ *float interval list* ⇒ *float list* ⇒ *floatarith* ⇒ *taylor-model option*
where *tm-floatarith prec ord I a f* = (
map-option (λcs .
 let (*pf*, *pi*) = *tmf-polys cs*;
 - = *compute-bound-tm prec* (*List.map2* (–) *I a*);
 e = *round-interval prec* (*Ipoly* (*List.map2* (–) *I a*) *pi*) — TODO: use *compute-bound-tm* here?!
 in *TaylorModel* *pf e*
) (*tmf-ivl-cs prec ord I a f*)
) — Compute a Taylor model from an arbitrary, univariate floatarith expression, if possible. This is used to compute Taylor models for elemental functions like sin, cos, exp, etc.

term *compute-bound-poly*

lemma *tmf-c-correct*:

fixes *A*::*float interval list* **and** *I*::*float interval* **and** *f*::*floatarith* **and** *a*::*real list*

assumes a all-in A
assumes $tmf\text{-}c$ prec A f $i = \text{Some } I$
shows $interpret\text{-}floatarith$ ($deriv\text{-}rec$ f i) a / $fact$ $i \in_r I$
 $\langle proof \rangle$

lemma $tmf\text{-}ivl\text{-}cs\text{-}length$:
assumes $tmf\text{-}ivl\text{-}cs$ prec n A a $f = \text{Some } cs$
shows $length$ $cs = n + 1$
 $\langle proof \rangle$

lemma $tmf\text{-}ivl\text{-}cs\text{-}correct$:
fixes $A::float$ interval list **and** $f::floatarith$
assumes a all-in I
assumes $tmf\text{-}ivl\text{-}cs$ prec ord I a $f = \text{Some } cs$
shows $\bigwedge i. i < ord \implies tmf\text{-}c$ prec (map interval-of a) f $i = \text{Some } (cs!i)$
and $tmf\text{-}c$ prec I f ord = $\text{Some } (cs!ord)$
and $length$ $cs = \text{Suc } ord$
 $\langle proof \rangle$

lemma $Ipoly\text{-}fst\text{-}tmf\text{-}polys$:
 $Ipoly$ xs (fst ($tmf\text{-}polys$ z)) = $(\sum_{i < length\ z} xs ! 0 \wedge i * (mid (z ! i)))$
for $xs::real$ list
 $\langle proof \rangle$

lemma $insertion\text{-}fst\text{-}tmf\text{-}polys$:
 $insertion$ e (fst ($tmf\text{-}polys$ z)) = $(\sum_{i < length\ z} e 0 \wedge i * (mid (z ! i)))$
for $e::nat \Rightarrow real$
 $\langle proof \rangle$

lemma $Ipoly\text{-}snd\text{-}tmf\text{-}polys$:
 $set\text{-}of$ ($horner\text{-}eval$ ($real\text{-}interval$ o centered o nth z) x ($length$ z)) \subseteq $set\text{-}of$ ($Ipoly$ [x] ($map\text{-}poly$ $real\text{-}interval$ (snd ($tmf\text{-}polys$ z))))
 $\langle proof \rangle$

lemma $zero\text{-}interval$ [$intro,simp$]: $0 \in_i 0$
 $\langle proof \rangle$

lemma $sum\text{-}in\text{-}intervalI$: sum f $X \in_i$ sum g X **if** $\bigwedge x. x \in X \implies f$ $x \in_i$ g x
for $f :: - \Rightarrow 'a :: ordered\text{-}comm\text{-}monoid\text{-}add$
 $\langle proof \rangle$

lemma $set\text{-}of\text{-}sum\text{-}subset$: $set\text{-}of$ (sum f X) \subseteq $set\text{-}of$ (sum g X)
if $\bigwedge x. x \in X \implies set\text{-}of$ (f x) \subseteq $set\text{-}of$ (g x)
for $f :: - \Rightarrow 'a::linordered\text{-}ab\text{-}group\text{-}add$ interval
 $\langle proof \rangle$

lemma $interval\text{-}of\text{-}plus$: $interval\text{-}of$ ($a + b$) = $interval\text{-}of$ a + $interval\text{-}of$ b
 $\langle proof \rangle$

lemma *interval-of-uminus*: $\text{interval-of } (- a) = - \text{interval-of } a$
 ⟨proof⟩

lemma *interval-of-zero*: $\text{interval-of } 0 = 0$
 ⟨proof⟩

lemma *interval-of-sum*: $\text{interval-of } (\text{sum } f X) = \text{sum } (\lambda x. \text{interval-of } (f x)) X$
 ⟨proof⟩

lemma *interval-of-prod*: $\text{interval-of } (a * b) = \text{interval-of } a * \text{interval-of } b$
 ⟨proof⟩

lemma *in-set-of-interval-of[simp]*: $x \in_i (\text{interval-of } y) \longleftrightarrow x = y \text{ for } x y :: 'a :: \text{order}$
 ⟨proof⟩

lemma *real-interval-Ipoly*: $\text{real-interval } (Ipoly \text{ } xs \text{ } p) = Ipoly (\text{map } \text{real-interval } xs)$
 (*map-poly real-interval p*)
if $\text{num-params } p \leq \text{length } xs$
 ⟨proof⟩

lemma *num-params-tmf-polys1*: $\text{num-params } (\text{fst } (tmf-polys \text{ } z)) \leq \text{Suc } 0$
 ⟨proof⟩

lemma *num-params-tmf-polys2*: $\text{num-params } (\text{snd } (tmf-polys \text{ } z)) \leq \text{Suc } 0$
 ⟨proof⟩

lemma *set-of-real-interval-subset*: $\text{set-of } (\text{real-interval } x) \subseteq \text{set-of } (\text{real-interval } y)$
if $\text{set-of } x \subseteq \text{set-of } y$
 ⟨proof⟩

theorem *tm-floatarith*:

assumes t : *tm-floatarith prec ord I xs f = Some t*
assumes a : *xs all-in I and x: x ∈_r I ! 0*
assumes $xs-ne$: $xs \neq []$
assumes $deriv$: $\bigwedge x. x \in_r I ! 0 \implies \text{isDERIV } 0 f (xs[0 := x])$
assumes i : $0 < i \implies i < \text{length } xs \implies e \text{ } i = \text{real-of-float } (xs ! i)$
assumes $diff-e$: $(x - \text{real-of-float } (xs ! 0)) = e \text{ } 0$
shows *interpret-floatarith f (xs[0:=x]) ∈_i range-tm e t*
 ⟨proof⟩

5.5 Operations on Taylor models

fun *tm-norm-poly* :: *taylor-model* \Rightarrow *taylor-model*

where *tm-norm-poly* (*TaylorModel* $p \text{ } e$) = *TaylorModel* (*polynome* p) e

— Normalizes the Taylor model by transforming its polynomial into horner form.

fun *tm-lower-order* *tm-lower-order-of-normed* :: *nat* \Rightarrow *float interval list* \Rightarrow *float interval list* \Rightarrow *taylor-model* \Rightarrow *taylor-model*

where *tm-lower-order prec ord I a t* = *tm-lower-order-of-normed prec ord I a*

(*tm-norm-poly t*)
 | *tm-lower-order-of-normed prec ord I a (TaylorModel p e) = (*
 let (l, r) = split-by-degree ord p
 in TaylorModel l (round-interval prec (e + compute-bound-poly prec r I a))
)

— Reduces the degree of a Taylor model’s polynomial to *n* and keeps it range by increasing the error bound.

fun *tm-round-floats tm-round-floats-of-normed :: nat ⇒ float interval list ⇒ float interval list ⇒ taylor-model ⇒ taylor-model*
where *tm-round-floats prec I a t = tm-round-floats-of-normed prec I a (tm-norm-poly t)*
 | *tm-round-floats-of-normed prec I a (TaylorModel p e) = (*
 let (l, r) = split-by-prec prec p
 in TaylorModel l (round-interval prec (e + compute-bound-poly prec r I a))
)

— Rounding of Taylor models. Rounds both the coefficients of the polynomial and the floats in the error bound.

fun *tm-norm tm-norm' :: nat ⇒ nat ⇒ float interval list ⇒ float interval list ⇒ taylor-model ⇒ taylor-model*
where *tm-norm prec ord I a t = tm-norm' prec ord I a (tm-norm-poly t)*
 | *tm-norm' prec ord I a t = tm-round-floats-of-normed prec I a (tm-lower-order-of-normed prec ord I a t)*

— Normalization of taylor models. Performs order lowering and rounding on taylor models, also converts the polynomial into horner form.

fun *tm-neg :: taylor-model ⇒ taylor-model*
where *tm-neg (TaylorModel p e) = TaylorModel (~_p p) (-e)*

fun *tm-add :: taylor-model ⇒ taylor-model ⇒ taylor-model*
where *tm-add (TaylorModel p1 e1) (TaylorModel p2 e2) = TaylorModel (p1 +_p p2) (e1 + e2)*

fun *tm-sub :: taylor-model ⇒ taylor-model ⇒ taylor-model*
where *tm-sub t1 t2 = tm-add t1 (tm-neg t2)*

fun *tm-mul :: nat ⇒ nat ⇒ float interval list ⇒ float interval list ⇒ taylor-model ⇒ taylor-model ⇒ taylor-model*
where *tm-mul prec ord I a (TaylorModel p1 e1) (TaylorModel p2 e2) = (*
 let d1 = compute-bound-poly prec p1 I a;
 d2 = compute-bound-poly prec p2 I a;
 *p = p1 *_p p2;*
 *e = e1*d2 + d1*e2 + e1*e2*
 in tm-norm' prec ord I a (TaylorModel p e)
)

lemmas [*simp del*] = *tm-norm'.simps*

fun *tm-pow :: nat ⇒ nat ⇒ float interval list ⇒ float interval list ⇒ taylor-model*

```

⇒ nat ⇒ taylor-model
  where tm-pow prec ord I a t 0 = tm-const 1
  | tm-pow prec ord I a t (Suc n) = (
    if odd (Suc n)
    then tm-mul prec ord I a t (tm-pow prec ord I a t n)
    else let t' = tm-pow prec ord I a t ((Suc n) div 2)
         in tm-mul prec ord I a t' t'
  )

```

Evaluates a float polynomial, using a Taylor model as the parameter. This is used to compose Taylor models.

```

fun eval-poly-at-tm :: nat ⇒ nat ⇒ float interval list ⇒ float interval list ⇒ float
poly ⇒ taylor-model ⇒ taylor-model
  where eval-poly-at-tm prec ord I a (poly.C c) t = tm-const c
  | eval-poly-at-tm prec ord I a (poly.Bound n) t = t
  | eval-poly-at-tm prec ord I a (poly.Add p1 p2) t
    = tm-add (eval-poly-at-tm prec ord I a p1 t)
              (eval-poly-at-tm prec ord I a p2 t)
  | eval-poly-at-tm prec ord I a (poly.Sub p1 p2) t
    = tm-sub (eval-poly-at-tm prec ord I a p1 t)
              (eval-poly-at-tm prec ord I a p2 t)
  | eval-poly-at-tm prec ord I a (poly.Mul p1 p2) t
    = tm-mul prec ord I a (eval-poly-at-tm prec ord I a p1 t)
                          (eval-poly-at-tm prec ord I a p2 t)
  | eval-poly-at-tm prec ord I a (poly.Neg p) t
    = tm-neg (eval-poly-at-tm prec ord I a p t)
  | eval-poly-at-tm prec ord I a (poly.Pw p n) t
    = tm-pow prec ord I a (eval-poly-at-tm prec ord I a p t) n
  | eval-poly-at-tm prec ord I a (poly.CN c n p) t = (
    let pt = eval-poly-at-tm prec ord I a p t;
        t-mul-pt = tm-mul prec ord I a t pt
    in tm-add (eval-poly-at-tm prec ord I a c t) t-mul-pt
  )

```

```

fun tm-inc-err :: float interval ⇒ taylor-model ⇒ taylor-model
  where tm-inc-err i (TaylorModel p e) = TaylorModel p (e + i)

```

```

fun tm-comp :: nat ⇒ nat ⇒ float interval list ⇒ float interval list ⇒ float ⇒
taylor-model ⇒ taylor-model ⇒ taylor-model
  where tm-comp prec ord I a ta (TaylorModel p e) t = (
    let t-sub-ta = tm-sub t (tm-const ta);
        pt = eval-poly-at-tm prec ord I a p t-sub-ta
    in tm-inc-err e pt
  )

```

tm-max, *tm-min* and *tm-abs* are implemented extremely naively, because I don't expect them to be very useful. But the implementation is fairly modular, i.e. *tm*-{*abs,min,max*} all can easily be swapped out, as long as the corresponding correctness lemmas *tm*-{*abs,min,max*}-range are updated

as well.

fun *tm-abs* :: nat \Rightarrow float interval list \Rightarrow float interval list \Rightarrow taylor-model \Rightarrow taylor-model

where *tm-abs* prec I a t = (
 let bound = compute-bound-tm prec I a t; abs-bound=Ivl (0::float) (max (abs (lower bound)) (abs (upper bound)))
 in TaylorModel (poly.C (mid abs-bound)) (centered abs-bound))

fun *tm-union* :: nat \Rightarrow float interval list \Rightarrow float interval list \Rightarrow taylor-model \Rightarrow taylor-model \Rightarrow taylor-model

where *tm-union* prec I a t1 t2 = (
 let b1 = compute-bound-tm prec I a t1; b2 = compute-bound-tm prec I a t2;
 b-combined = sup b1 b2
 in TaylorModel (poly.C (mid b-combined)) (centered b-combined))

fun *tm-min* :: nat \Rightarrow float interval list \Rightarrow float interval list \Rightarrow taylor-model \Rightarrow taylor-model \Rightarrow taylor-model

where *tm-min* prec I a t1 t2 = *tm-union* prec I a t1 t2

fun *tm-max* :: nat \Rightarrow float interval list \Rightarrow float interval list \Rightarrow taylor-model \Rightarrow taylor-model \Rightarrow taylor-model

where *tm-max* prec I a t1 t2 = *tm-union* prec I a t1 t2

Rangeity of is preserved by our operations on Taylor models.

lemma *insertion-polyadd[simp]*: insertion e (a +_p b) = insertion e a + insertion e b

for a b::'a::ring-1 poly
 <proof>

lemma *insertion-polyneg[simp]*: insertion e (\sim_p b) = - insertion e b

for b::'a::ring-1 poly
 <proof>

lemma *insertion-polysub[simp]*: insertion e (a -_p b) = insertion e a - insertion e b

for a b::'a::ring-1 poly
 <proof>

lemma *insertion-polymul[simp]*: insertion e (a *_p b) = insertion e a * insertion e b

for a b::'a::comm-ring-1 poly
 <proof>

lemma *insertion-polypow[simp]*: insertion e (a $\hat{^}_p$ b) = insertion e a $\hat{^}$ b

for a::'a::comm-ring-1 poly
 <proof>

lemma *insertion-polynate [simp]*:

insertion bs (polynate p) = (insertion bs p :: 'a::comm-ring-1)
<proof>

lemma *tm-norm-poly-range:*
assumes $x \in_i \text{range-tm } e \ t$
shows $x \in_i \text{range-tm } e \ (\text{tm-norm-poly } t)$
<proof>

lemma *split-by-degree-correct-insertion:*
fixes $x :: \text{nat} \Rightarrow \text{real}$ **and** $p :: \text{float poly}$
assumes *split-by-degree ord p = (l, r)*
shows $\text{maxdegree } l \leq \text{ord}$ (**is** ?P1)
and $\text{insertion } x \ p = \text{insertion } x \ l + \text{insertion } x \ r$ (**is** ?P2)
and $\text{num-params } l \leq \text{num-params } p$ (**is** ?P3)
and $\text{num-params } r \leq \text{num-params } p$ (**is** ?P4)
<proof>

lemma *split-by-prec-correct-insertion:*
fixes $x :: \text{nat} \Rightarrow \text{real}$ **and** $p :: \text{float poly}$
assumes *split-by-prec ord p = (l, r)*
shows $\text{insertion } x \ p = \text{insertion } x \ l + \text{insertion } x \ r$ (**is** ?P1)
and $\text{num-params } l \leq \text{num-params } p$ (**is** ?P2)
and $\text{num-params } r \leq \text{num-params } p$ (**is** ?P3)
<proof>

lemma *tm-lower-order-of-normed-range:*
assumes $x \in_i \text{range-tm } e \ t$
assumes *dev: develops-at-within e a I*
assumes $\text{num-params } (\text{tm-poly } t) \leq \text{length } I$
shows $x \in_i \text{range-tm } e \ (\text{tm-lower-order-of-normed prec ord } I \ a \ t)$
<proof>

lemma *num-params-tm-norm-poly-le: num-params (tm-poly (tm-norm-poly t)) ≤ X*
if $\text{num-params } (\text{tm-poly } t) \leq X$
<proof>

lemma *tm-lower-order-range:*
assumes $x \in_i \text{range-tm } e \ t$
assumes *dev: develops-at-within e a I*
assumes $\text{num-params } (\text{tm-poly } t) \leq \text{length } I$
shows $x \in_i \text{range-tm } e \ (\text{tm-lower-order prec ord } I \ a \ t)$
<proof>

lemma *tm-round-floats-of-normed-range:*
assumes $x \in_i \text{range-tm } e \ t$
assumes *dev: develops-at-within e a I*
assumes $\text{num-params } (\text{tm-poly } t) \leq \text{length } I$
shows $x \in_i \text{range-tm } e \ (\text{tm-round-floats-of-normed prec } I \ a \ t)$

— TODO: this is a clone of $[[?x \in_i \text{range-tm } ?e ?t; \text{develops-at-within } ?e ?a ?I;$
 $\text{num-params } (tm\text{-poly } ?t) \leq \text{length } ?I]] \implies ?x \in_i \text{range-tm } ?e (tm\text{-lower-order-of-normed}$
 $?prec ?ord ?I ?a ?t) \rightarrow$ general sweeping method!
 ⟨proof⟩

lemma *num-params-split-by-degree-le*: $\text{num-params } (fst (split\text{-by-degree } ord\ x)) \leq$
 K
 $\text{num-params } (snd (split\text{-by-degree } ord\ x)) \leq K$
if $\text{num-params } x \leq K$ **for** $x::\text{float poly}$
 ⟨proof⟩

lemma *num-params-split-by-prec-le*: $\text{num-params } (fst (split\text{-by-prec } ord\ x)) \leq K$
 $\text{num-params } (snd (split\text{-by-prec } ord\ x)) \leq K$
if $\text{num-params } x \leq K$ **for** $x::\text{float poly}$
 ⟨proof⟩

lemma *num-params-tm-norm'-le*:
 $\text{num-params } (tm\text{-poly } (tm\text{-round-floats-of-normed } prec\ I\ a\ t)) \leq X$
if $\text{num-params } (tm\text{-poly } t) \leq X$
 ⟨proof⟩

lemma *tm-round-floats-range*:
assumes $x \in_i \text{range-tm } e\ t$ *develops-at-within* $e\ a\ I$ $\text{num-params } (tm\text{-poly } t) \leq$
 $\text{length } I$
shows $x \in_i \text{range-tm } e (tm\text{-round-floats } prec\ I\ a\ t)$
 ⟨proof⟩

lemma *num-params-tm-lower-order-of-normed-le*: $\text{num-params } (tm\text{-poly } (tm\text{-lower-order-of-normed}$
 $prec\ ord\ I\ a\ t)) \leq X$
if $\text{num-params } (tm\text{-poly } t) \leq X$
 ⟨proof⟩

lemma *tm-norm'-range*:
assumes $x \in_i \text{range-tm } e\ t$ *develops-at-within* $e\ a\ I$ $\text{num-params } (tm\text{-poly } t) \leq$
 $\text{length } I$
shows $x \in_i \text{range-tm } e (tm\text{-norm}'\ prec\ ord\ I\ a\ t)$
 ⟨proof⟩

lemma *num-params-tm-norm'*:
 $\text{num-params } (tm\text{-poly } (tm\text{-norm}'\ prec\ ord\ I\ a\ t)) \leq X$
if $\text{num-params } (tm\text{-poly } t) \leq X$
 ⟨proof⟩

lemma *tm-norm-range*:
assumes $x \in_i \text{range-tm } e\ t$ *develops-at-within* $e\ a\ I$ $\text{num-params } (tm\text{-poly } t) \leq$
 $\text{length } I$
shows $x \in_i \text{range-tm } e (tm\text{-norm } prec\ ord\ I\ a\ t)$
 ⟨proof⟩

lemmas [*simp del*] = *tm-norm.simps*

lemma *tm-neg-range*:

assumes $x \in_i \text{range-tm } e \ t$

shows $-x \in_i \text{range-tm } e \ (\text{tm-neg } t)$

<proof>

lemmas [*simp del*] = *tm-neg.simps*

lemma *tm-bound-tm-add[*simp*]*: $\text{tm-bound } (\text{tm-add } t1 \ t2) = \text{tm-bound } t1 + \text{tm-bound } t2$

<proof>

lemma *interval-of-add*: $\text{interval-of } (a + b) = \text{interval-of } a + \text{interval-of } b$

<proof>

lemma *tm-add-range*:

$x + y \in_i \text{range-tm } e \ (\text{tm-add } t1 \ t2)$

if $x \in_i \text{range-tm } e \ t1$

$y \in_i \text{range-tm } e \ t2$

<proof>

lemmas [*simp del*] = *tm-add.simps*

lemma *tm-sub-range*:

assumes $x \in_i \text{range-tm } e \ t1$

assumes $y \in_i \text{range-tm } e \ t2$

shows $x - y \in_i \text{range-tm } e \ (\text{tm-sub } t1 \ t2)$

<proof>

lemmas [*simp del*] = *tm-sub.simps*

lemma *set-of-intervalI*: $\text{set-of } (\text{interval-of } y) \subseteq \text{set-of } Y$ **if** $y \in_i Y$ **for** $y::'a::\text{order}$

<proof>

lemma *set-of-real-intervalI*: $\text{set-of } (\text{interval-of } y) \subseteq \text{set-of } (\text{real-interval } Y)$ **if** $y \in_r Y$

<proof>

lemma *tm-mul-range*:

assumes $x \in_i \text{range-tm } e \ t1$

assumes $y \in_i \text{range-tm } e \ t2$

assumes *dev*: *develops-at-within* $e \ a \ I$

assumes *params*: $\text{num-params } (\text{tm-poly } t1) \leq \text{length } I \ \text{num-params } (\text{tm-poly } t2) \leq \text{length } I$

shows $x * y \in_i \text{range-tm } e \ (\text{tm-mul } \text{prec } \text{ord } I \ a \ t1 \ t2)$

<proof>

lemma *num-params-tm-mul-le*:

$\text{num-params } (\text{tm-poly } (\text{tm-mul } \text{prec } \text{ord } I \ a \ t1 \ t2)) \leq X$

if $\text{num-params } (\text{tm-poly } t1) \leq X$

$num\text{-}params (tm\text{-}poly t2) \leq X$
 ⟨proof⟩

lemmas [*simp del*] = *tm-pow.simps*— TODO: make a systematic decision

lemma

shows *tm-pow-range*: $num\text{-}params (tm\text{-}poly t) \leq length\ I \implies$
 $develops\text{-}at\text{-}within\ e\ a\ I \implies$
 $x \in_i\ range\text{-}tm\ e\ t \implies$
 $x \hat{\ } n \in_i\ range\text{-}tm\ e\ (tm\text{-}pow\ prec\ ord\ I\ a\ t\ n)$
and *num-params-tm-pow-le*[*THEN order-trans*]:
 $num\text{-}params (tm\text{-}poly (tm\text{-}pow\ prec\ ord\ I\ a\ t\ n)) \leq num\text{-}params (tm\text{-}poly\ t)$
 ⟨proof⟩

lemma *num-params-tm-add-le*:

$num\text{-}params (tm\text{-}poly (tm\text{-}add\ t1\ t2)) \leq X$
if $num\text{-}params (tm\text{-}poly\ t1) \leq X$
 $num\text{-}params (tm\text{-}poly\ t2) \leq X$
 ⟨proof⟩

lemma *num-params-tm-neg-eq*[*simp*]:

$num\text{-}params (tm\text{-}poly (tm\text{-}neg\ t1)) = num\text{-}params (tm\text{-}poly\ t1)$
 ⟨proof⟩

lemma *num-params-tm-sub-le*:

$num\text{-}params (tm\text{-}poly (tm\text{-}sub\ t1\ t2)) \leq X$
if $num\text{-}params (tm\text{-}poly\ t1) \leq X$
 $num\text{-}params (tm\text{-}poly\ t2) \leq X$
 ⟨proof⟩

lemma *num-params-eval-poly-le*: $num\text{-}params (tm\text{-}poly (eval\text{-}poly\text{-}at\text{-}tm\ prec\ ord\ I\ a\ p\ t)) \leq x$

if $num\text{-}params (tm\text{-}poly\ t) \leq x\ num\text{-}params\ p \leq max\ 1\ x$
 ⟨proof⟩

lemma *eval-poly-at-tm-range*:

assumes $num\text{-}params\ p \leq 1$
assumes *tg-def*: $e' 0 \in_i\ range\text{-}tm\ e\ tg$
assumes *dev*: $develops\text{-}at\text{-}within\ e\ a\ I$ **and** *params*: $num\text{-}params (tm\text{-}poly\ tg) \leq$
 $length\ I$
shows *insertion* $e' p \in_i\ range\text{-}tm\ e\ (eval\text{-}poly\text{-}at\text{-}tm\ prec\ ord\ I\ a\ p\ tg)$
 ⟨proof⟩

lemma *tm-inc-err-range*: $x \in_i\ range\text{-}tm\ e\ (tm\text{-}inc\text{-}err\ i\ t)$

if $x \in_i\ range\text{-}tm\ e\ t + real\text{-}interval\ i$
 ⟨proof⟩

lemma *num-params-tm-inc-err*: $num\text{-}params (tm\text{-}poly (tm\text{-}inc\text{-}err\ i\ t)) \leq X$

if $num\text{-}params (tm\text{-}poly\ t) \leq X$

<proof>

lemma *num-params-tm-comp-le*: $\text{num-params } (tm\text{-poly } (tm\text{-comp } prec \text{ ord } I \ a \ ga \ tf \ tg)) \leq X$

if $\text{num-params } (tm\text{-poly } tf) \leq \max 1 \ X \ \text{num-params } (tm\text{-poly } tg) \leq X$
<proof>

lemma *tm-comp-range*:

assumes *tf-def*: $x \in_i \text{range-tm } e' \ tf$

assumes *tg-def*: $e' \ 0 \in_i \text{range-tm } e \ (tm\text{-sub } tg \ (tm\text{-const } ga))$

assumes *params*: $\text{num-params } (tm\text{-poly } tf) \leq 1 \ \text{num-params } (tm\text{-poly } tg) \leq \text{length } I$

assumes *dev*: *develops-at-within* $e \ a \ I$

shows $x \in_i \text{range-tm } e \ (tm\text{-comp } prec \ \text{ord } I \ a \ ga \ tf \ tg)$

<proof>

lemma *mid-centered-collapse*:

$\text{interval-of } (real\text{-of-float } (mid \ abs\text{-bound})) + \text{real-interval } (centered \ abs\text{-bound}) =$
 $\text{real-interval } abs\text{-bound}$

<proof>

lemmas [*simp del*] = *tm-abs.simps*

lemma *tm-abs-range*:

assumes *x*: $x \in_i \text{range-tm } e \ t$

assumes *n*: $\text{num-params } (tm\text{-poly } t) \leq \text{length } I$ **and** *d*: *develops-at-within* $e \ a \ I$

shows $abs \ x \in_i \text{range-tm } e \ (tm\text{-abs } prec \ I \ a \ t)$

<proof>

lemma *num-params-tm-abs-le*: $\text{num-params } (tm\text{-poly } (tm\text{-abs } prec \ I \ a \ t)) \leq X$ **if**
 $\text{num-params } (tm\text{-poly } t) \leq X$

<proof>

lemma *real-interval-sup*: $\text{real-interval } (sup \ a \ b) = sup \ (\text{real-interval } a) \ (\text{real-interval } b)$

<proof>

lemma *in-interval-supI1*: $x \in_i \ a \implies x \in_i \ sup \ a \ b$

and *in-interval-supI2*: $x \in_i \ b \implies x \in_i \ sup \ a \ b$

for $x::'a::lattice$

<proof>

lemma *tm-union-range-left*:

assumes $x \in_i \text{range-tm } e \ t1$

$\text{num-params } (tm\text{-poly } t1) \leq \text{length } I$ *develops-at-within* $e \ a \ I$

shows $x \in_i \text{range-tm } e \ (tm\text{-union } prec \ I \ a \ t1 \ t2)$

<proof>

lemma *tm-union-range-right*:

assumes $x \in_i \text{range-tm } e \ t2$

$num\text{-}params (tm\text{-}poly\ t2) \leq length\ I\ develops\text{-}at\text{-}within\ e\ a\ I$
shows $x \in_i\ range\text{-}tm\ e\ (tm\text{-}union\ prec\ I\ a\ t1\ t2)$
 ⟨proof⟩

lemma *num-params-tm-union-le*:

$num\text{-}params (tm\text{-}poly (tm\text{-}union\ prec\ I\ a\ t1\ t2)) \leq X$
if $num\text{-}params (tm\text{-}poly\ t1) \leq X\ num\text{-}params (tm\text{-}poly\ t2) \leq X$
 ⟨proof⟩

lemmas [*simp del*] = *tm-union.simps tm-min.simps tm-max.simps*

lemma *tm-min-range*:

assumes $x \in_i\ range\text{-}tm\ e\ t1$
assumes $y \in_i\ range\text{-}tm\ e\ t2$
 $num\text{-}params (tm\text{-}poly\ t1) \leq length\ I$
 $num\text{-}params (tm\text{-}poly\ t2) \leq length\ I$
 $develops\text{-}at\text{-}within\ e\ a\ I$
shows $min\ x\ y \in_i\ range\text{-}tm\ e\ (tm\text{-}min\ prec\ I\ a\ t1\ t2)$
 ⟨proof⟩

lemma *tm-max-range*:

assumes $x \in_i\ range\text{-}tm\ e\ t1$
assumes $y \in_i\ range\text{-}tm\ e\ t2$
 $num\text{-}params (tm\text{-}poly\ t1) \leq length\ I$
 $num\text{-}params (tm\text{-}poly\ t2) \leq length\ I$
 $develops\text{-}at\text{-}within\ e\ a\ I$
shows $max\ x\ y \in_i\ range\text{-}tm\ e\ (tm\text{-}max\ prec\ I\ a\ t1\ t2)$
 ⟨proof⟩

5.6 Computing Taylor models for multivariate expressions

Compute Taylor models for expressions of the form "f (g x)", where f is an elementary function like exp or cos, by composing Taylor models for f and g. For our correctness proof, we need to make it explicit that the range of g on I is inside the domain of f, by introducing the *f-exists-on* predicate.

```

fun compute-tm-by-comp :: nat ⇒ nat ⇒ float interval list ⇒ float interval list ⇒
floatarith ⇒ taylor-model option ⇒ (float interval ⇒ bool) ⇒ taylor-model option
  where compute-tm-by-comp prec ord I a f g f-exists-on = (
    case g
    of Some tg ⇒ (
      let gI = compute-bound-tm prec I a tg;
      ga = mid (compute-bound-tm prec a a tg)
      in if f-exists-on gI
      then map-option (λtf. tm-comp prec ord I a ga tf tg) (tm-floatarith prec
ord [gI] [ga] f)
      else None)
    | - ⇒ None
  )
  
```

Compute Taylor models with numerical precision $prec$ of degree ord , with Taylor models in the environment env whose variables are jointly interpreted with domain I and expanded around point a . from floatarith expressions on a rectangular domain.

fun *approx-tm* :: *nat* \Rightarrow *nat* \Rightarrow *float interval list* \Rightarrow *float interval list* \Rightarrow *floatarith* \Rightarrow *taylor-model list* \Rightarrow *taylor-model option*

where *approx-tm* - - *I* - (*Num* *c*) *env* = *Some* (*tm-const* *c*)
| *approx-tm* - - *I* *a* (*Var* *n*) *env* = (if *n* < *length env* then *Some* (*env* ! *n*) else *None*)
| *approx-tm* *prec ord I a* (*Add* *l r*) *env* = (
 case (*approx-tm* *prec ord I a l env*, *approx-tm* *prec ord I a r env*)
 of (*Some* *t1*, *Some* *t2*) \Rightarrow *Some* (*tm-add* *t1 t2*)
 | - \Rightarrow *None*)
| *approx-tm* *prec ord I a* (*Minus* *f*) *env*
 = *map-option* *tm-neg* (*approx-tm* *prec ord I a f env*)
| *approx-tm* *prec ord I a* (*Mult* *l r*) *env* = (
 case (*approx-tm* *prec ord I a l env*, *approx-tm* *prec ord I a r env*)
 of (*Some* *t1*, *Some* *t2*) \Rightarrow *Some* (*tm-mul* *prec ord I a t1 t2*)
 | - \Rightarrow *None*)
| *approx-tm* *prec ord I a* (*Power* *f k*) *env*
 = *map-option* (λt . *tm-pow* *prec ord I a t k*)
 (*approx-tm* *prec ord I a f env*)
| *approx-tm* *prec ord I a* (*Inverse* *f*) *env*
 = *compute-tm-by-comp* *prec ord I a* (*Inverse* (*Var* 0)) (*approx-tm* *prec ord I a f env*) (λx . 0 < *lower* *x* \vee *upper* *x* < 0)
| *approx-tm* *prec ord I a* (*Cos* *f*) *env*
 = *compute-tm-by-comp* *prec ord I a* (*Cos* (*Var* 0)) (*approx-tm* *prec ord I a f env*) (λx . *True*)
| *approx-tm* *prec ord I a* (*Arctan* *f*) *env*
 = *compute-tm-by-comp* *prec ord I a* (*Arctan* (*Var* 0)) (*approx-tm* *prec ord I a f env*) (λx . *True*)
| *approx-tm* *prec ord I a* (*Exp* *f*) *env*
 = *compute-tm-by-comp* *prec ord I a* (*Exp* (*Var* 0)) (*approx-tm* *prec ord I a f env*) (λx . *True*)
| *approx-tm* *prec ord I a* (*Ln* *f*) *env*
 = *compute-tm-by-comp* *prec ord I a* (*Ln* (*Var* 0)) (*approx-tm* *prec ord I a f env*) (λx . 0 < *lower* *x*)
| *approx-tm* *prec ord I a* (*Sqrt* *f*) *env*
 = *compute-tm-by-comp* *prec ord I a* (*Sqrt* (*Var* 0)) (*approx-tm* *prec ord I a f env*) (λx . 0 < *lower* *x*)
| *approx-tm* *prec ord I a* *Pi* *env* = *Some* (*tm-pi* *prec*)
| *approx-tm* *prec ord I a* (*Abs* *f*) *env*
 = *map-option* (*tm-abs* *prec I a*) (*approx-tm* *prec ord I a f env*)
| *approx-tm* *prec ord I a* (*Min* *l r*) *env* = (
 case (*approx-tm* *prec ord I a l env*, *approx-tm* *prec ord I a r env*)
 of (*Some* *t1*, *Some* *t2*) \Rightarrow *Some* (*tm-min* *prec I a t1 t2*)
 | - \Rightarrow *None*)
| *approx-tm* *prec ord I a* (*Max* *l r*) *env* = (
 case (*approx-tm* *prec ord I a l env*, *approx-tm* *prec ord I a r env*)
 of (*Some* *t1*, *Some* *t2*) \Rightarrow *Some* (*tm-max* *prec I a t1 t2*)
 | - \Rightarrow *None*)

case (approx-tm prec ord I a l env, approx-tm prec ord I a r env)
 of (Some t1, Some t2) \Rightarrow Some (tm-max prec I a t1 t2)
 | - \Rightarrow None
 | approx-tm prec ord I a (Powr l r) env = None — TODO
 | approx-tm prec ord I a (Floor l) env = None — TODO

lemma mid-in-real-interval: mid $i \in_r i$
 ⟨proof⟩

lemma set-of-real-interval-mono:set-of (real-interval x) \subseteq set-of (real-interval y)
 if set-of x \subseteq set-of y
 ⟨proof⟩

lemmas [simp del] = compute-bound-poly.simps tm-floatarith.simps

lemmas [simp del] = tmf-ivl-cs.simps compute-bound-tm.simps tmf-polys.simps

lemma tm-floatarith-eq-Some-num-params:
 tm-floatarith prec ord a b f = Some tf \implies num-params (tm-poly tf) \leq 1
 ⟨proof⟩

lemma compute-tm-by-comp-range:
assumes max-Var-floatarith f \leq 1
assumes a: a all-subset I
assumes tx-range: $x \in_i$ range-tm e tg
assumes t-def: compute-tm-by-comp prec ord I a f (Some tg) c = Some t
assumes f-deriv:
 $\bigwedge x. x \in_r$ compute-bound-tm prec I a tg \implies c (compute-bound-tm prec I a tg)
 \implies isDERIV 0 f [x]
assumes params: num-params (tm-poly tg) \leq length I
and dev: develops-at-within e a I
shows interpret-floatarith f [x] \in_i range-tm e t
 ⟨proof⟩

lemmas [simp del] = compute-tm-by-comp.simps

lemma compute-tm-by-comp-num-params-le:
assumes compute-tm-by-comp prec ord I a f (Some t0) i = Some t
assumes $1 \leq X$ num-params (tm-poly t0) \leq X
shows num-params (tm-poly t) \leq X
 ⟨proof⟩

lemma compute-tm-by-comp-eq-Some-iff: compute-tm-by-comp prec ord I a f t0 i
 = Some t \iff
 ($\exists z x2. t0 =$ Some x2 \wedge
 tm-floatarith prec ord [compute-bound-tm prec I a x2]
 [mid (compute-bound-tm prec a a x2)] f =

Some z
 \wedge *tm-comp prec ord I a*
 $(\text{mid } (\text{compute-bound-tm } \text{prec } a \ a \ x2)) \ z \ x2 = t$
 \wedge *i (compute-bound-tm prec I a x2)*
<proof>

lemma *num-params-approx-tm:*
assumes *approx-tm prec ord I a f env = Some t*
assumes $\wedge \text{tm. } \text{tm} \in \text{set } \text{env} \implies \text{num-params } (\text{tm-poly } \text{tm}) \leq \text{length } I$
shows $\text{num-params } (\text{tm-poly } t) \leq \text{length } I$
<proof>

lemma *in-interval-realI: a \in_i I if a \in_r I <proof>*

lemma *all-subset-all-inI: map interval-of a all-subset I if a all-in I <proof>*

lemma *compute-tm-by-comp-None: compute-tm-by-comp p ord I a x None k = None <proof>*

lemma *approx-tm-num-Vars-None:*
assumes *max-Var-floatarith f > length env*
shows *approx-tm p ord I a f env = None <proof>*

lemma *approx-tm-num-Vars:*
assumes *approx-tm prec ord I a f env = Some t*
shows *max-Var-floatarith f \leq length env <proof>*

definition *range-tms e xs = map (range-tm e) xs*

lemma *approx-tm-range:*
assumes *a: a all-subset I*
assumes *t-def: approx-tm prec ord I a f env = Some t*
assumes *allin: xs all-in_i range-tms e env*
assumes *devs: develops-at-within e a I*
assumes *env: $\wedge \text{tm. } \text{tm} \in \text{set } \text{env} \implies \text{num-params } (\text{tm-poly } \text{tm}) \leq \text{length } I$*
shows *interpret-floatarith f xs \in_i range-tm e t <proof>*

Evaluate expression with Taylor models in environment.

5.7 Computing bounds for floatarith expressions

TODO: compare parametrization of input vs. uncertainty for input...

definition *tm-of-ivl-par n ivl = TaylorModel (CN (C ((upper ivl + lower ivl)*Float 1 (-1))) n*

($C ((upper\ ivl - lower\ ivl)*Float\ 1\ (-1)))\ 0$
 — track uncertainty in parameter n , which is to be interpreted over standardized domain $[-1, 1]$.

value $tm\text{-of-}ivl\text{-par}\ 3\ (Ivl\ (-1)\ 1)$

definition $tms\text{-of-}ivls\ ivls = map\ (\lambda(i, ivl). tm\text{-of-}ivl\text{-par}\ i\ ivl)\ (zip\ [0..<length\ ivls]\ ivls)$

value $tms\text{-of-}ivls\ [Ivl\ 1\ 2, Ivl\ 4\ 5]$

primrec $approx\text{-slp}'::nat \Rightarrow nat \Rightarrow float\ interval\ list \Rightarrow float\ interval\ list \Rightarrow slp \Rightarrow taylor\text{-model}\ list \Rightarrow taylor\text{-model}\ list\ option$

where

$approx\text{-slp}'\ p\ ord\ I\ a\ []\ xs = Some\ xs$
 $| approx\text{-slp}'\ p\ ord\ I\ a\ (ea\ \#\ eas)\ xs =$
 $\quad do\ \{$
 $\quad\quad r \leftarrow approx\text{-tm}\ p\ ord\ I\ a\ ea\ xs;$
 $\quad\quad approx\text{-slp}'\ p\ ord\ I\ a\ eas\ (r\ \#\ xs)$
 $\quad\ \}$

lemma $mem\text{-range-}tms\text{-Cons-iff}[simp]: x\ \#\ xs\ all\text{-in}_i\ range\text{-tms}\ e\ (X\ \#\ XS) \longleftrightarrow x \in_i\ range\text{-tm}\ e\ X \wedge xs\ all\text{-in}_i\ range\text{-tms}\ e\ XS$
 $\langle proof \rangle$

lemma $approx\text{-slp}'\text{-range}:$

assumes $i: i\ all\text{-subset}\ I$
assumes $dev: devlops\text{-at-}within\ e\ i\ I$
assumes $vs: vs\ all\text{-in}_i\ range\text{-tms}\ e\ VS\ (\bigwedge tm. tm \in set\ VS \implies num\text{-params}\ (tm\text{-poly}\ tm) \leq length\ I)$
assumes $appr: approx\text{-slp}'\ p\ ord\ I\ i\ ra\ VS = Some\ X$
shows $interpret\text{-slp}\ ra\ vs\ all\text{-in}_i\ range\text{-tms}\ e\ X$
 $\langle proof \rangle$

definition $approx\text{-slp}::nat \Rightarrow nat \Rightarrow nat \Rightarrow slp \Rightarrow taylor\text{-model}\ list \Rightarrow taylor\text{-model}\ list\ option$

where

$approx\text{-slp}\ p\ ord\ d\ slp\ tms =$
 $map\text{-option}\ (take\ d)$
 $(approx\text{-slp}'\ p\ ord\ (replicate\ (length\ tms)\ (Ivl\ (-1)\ 1))\ (replicate\ (length\ tms)\ 0)\ slp\ tms)$

lemma $length\text{-range-}tms[simp]: length\ (range\text{-tms}\ e\ VS) = length\ VS$
 $\langle proof \rangle$

lemma $set\text{-of-}Ivl: set\text{-of}\ (Ivl\ a\ b) = \{a .. b\}$ **if** $a \leq b$
 $\langle proof \rangle$

lemma $set\text{-of-}zero[simp]: set\text{-of}\ 0 = \{0::'a::ordered\text{-comm-}monoid\text{-add}\}$

```

  <proof>

theorem approx-slp-range-tms:
  assumes approx-slp p ord d slp VS = Some X
  assumes slp-def: slp = slp-of-fas fas
  assumes d-def: d = length fas
  assumes e: e ∈ UNIV → {-1 .. 1}
  assumes vs: vs all-ini range-tms e VS
  assumes lens:  $\bigwedge tm. tm \in \text{set } VS \implies \text{num-params } (tm\text{-poly } tm) \leq \text{length } vs$ 
  shows interpret-floatariths fas vs all-ini range-tms e X
  <proof>

end

end

theory Experiments
  imports Taylor-Models
  Affine-Arithmetic.Affine-Arithmetic
begin

instantiation interval::({show, preorder}) show begin

context includes interval.lifting begin
lift-definition shows-prec-interval::
  nat ⇒ 'a interval ⇒ char list ⇒ char list
  is  $\lambda p\ ivl\ s. (\text{shows-string } "Interval" \circ \text{shows } ivl) \ s$  <proof>

lift-definition shows-list-interval::
  'a interval list ⇒ char list ⇒ char list
  is  $\lambda ivls\ s. \text{shows-list } ivls\ s$  <proof>

instance
  <proof>
end

end

definition split-largest-interval :: float interval list ⇒ float interval list × float interval list where
split-largest-interval xs = (case sort-key (uminus o snd) (zip [0..<length xs] (map
  ( $\lambda x. \text{upper } x - \text{lower } x$ ) xs)) of Nil ⇒ ([], []))
  | (i, -)#- ⇒ let x = xs! i in (xs[i:=Ivl (lower x) ((upper x + lower x)*Float 1
  (-1))],
   $xs[i:=Ivl ((upper\ x + lower\ x)*Float\ 1\ (-1))\ (upper\ x)]$ ))

definition Inf-tm p params tm =
  lower (compute-bound-tm p (replicate params (Ivl (-1) (1))) (replicate params
  (Ivl 0 0)) tm)

```

```

primrec prove-pos::bool ⇒ nat ⇒ nat ⇒ nat ⇒
  (nat ⇒ nat ⇒ taylor-model list ⇒ taylor-model option) ⇒ float interval list list
⇒ bool where
  prove-pos prnt 0 p ord F X = (let - = if prnt then print (STR "# depth limit
exceeded↔") else () in False)
| prove-pos prnt (Suc i) p ord F XXS =
  (case XXS of [] ⇒ True | (X#XS) ⇒
  let
    params = length X;
    R = F p ord (tms-of-ivls X);
    - = if prnt then print (String.implode ((shows "# " o shows (map (λivl.
(lower ivl, upper ivl)) X)) "↔")) else ()
  in
    if R ≠ None ∧ 0 < Inf-tm p params (the R)
    then let - = if prnt then print (STR "# Success↔") else () in prove-pos prnt
i p ord F XS
    else let - = if prnt then print (String.implode ((shows "# Split (" o shows
((map-option (Inf-tm p params)) R) o shows "'") "↔")) else () in case split-largest-interval
X of (a, b) ⇒
      prove-pos prnt i p ord F (a#b#XS))

```

hide-const (open) prove-pos-slp

definition prove-pos-slp prnt prec ord fa i xs = (let slp = slp-of-fas [fa] in prove-pos
prnt i prec ord (λp ord xs.
case approx-slp prec ord 1 slp xs of None ⇒ None | Some [x] ⇒ Some x | Some
- ⇒ None) xs)

experiment begin

unbundle floatarith-notation

abbreviation schwefel ≡

$$(5.8806 / 10^{10}) + (\text{Var } 0 - (\text{Var } 1)^e)^e + (\text{Var } 1 - 1)^e + (\text{Var } 0 - (\text{Var } 2)^e)^e + (\text{Var } 2 - 1)^e$$

lemma prove-pos-slp True 30 0 schwefel 100000 [replicate 3 (Ivl (-10) 10)]
{proof}

abbreviation delta6 ≡ (Var 0 * Var 3 * (-Var 0 + Var 1 + Var 2 - Var 3 +
Var 4 + Var 5) +

$$\begin{aligned} & \text{Var } 1 * \text{Var } 4 * (\text{Var } 0 - \text{Var } 1 + \text{Var } 2 + \text{Var } 3 - \text{Var } 4 + \text{Var } 5) + \\ & \text{Var } 2 * \text{Var } 5 * (\text{Var } 0 + \text{Var } 1 - \text{Var } 2 + \text{Var } 3 + \text{Var } 4 - \text{Var } 5) \\ & - \text{Var } 1 * \text{Var } 2 * \text{Var } 3 \\ & - \text{Var } 0 * \text{Var } 2 * \text{Var } 4 \\ & - \text{Var } 0 * \text{Var } 1 * \text{Var } 5 \\ & - \text{Var } 3 * \text{Var } 4 * \text{Var } 5 \end{aligned}$$

lemma *prove-pos-slp True 30 3 delta6 10000 [replicate 6 (Ivl 4 (Float 104045 (-14)))]*
 ⟨proof⟩

abbreviation *caprasse* $\equiv (3.1801 + - \text{Var } 0 * (\text{Var } 2) \hat{e} 3 + 4 * \text{Var } 1 * (\text{Var } 2) \hat{e} 2 * \text{Var } 3 +$
 $4 * \text{Var } 0 * \text{Var } 2 * (\text{Var } 3) \hat{e} 2 + 2 * \text{Var } 1 * (\text{Var } 3) \hat{e} 3 + 4 * \text{Var } 0 * \text{Var } 2 + 4 * (\text{Var } 2) \hat{e} 2 - 10 * \text{Var } 1 * \text{Var } 3 +$
 $- 10 * (\text{Var } 3) \hat{e} 2 + 2)$

lemma *prove-pos-slp True 30 2 caprasse 10000 [replicate 4 (Ivl (-Float 1 (-1)) (Float 1 (-1)))]*
 ⟨proof⟩

abbreviation *magnetism* \equiv
 $0.25001 + (\text{Var } 0) \hat{e} 2 + 2 * (\text{Var } 1) \hat{e} 2 + 2 * (\text{Var } 2) \hat{e} 2 + 2 * (\text{Var } 3) \hat{e} 2 +$
 $2 * (\text{Var } 4) \hat{e} 2 + 2 * (\text{Var } 5) \hat{e} 2 +$
 $2 * (\text{Var } 6) \hat{e} 2 - \text{Var } 0$

end

end